Compositional Specification and Model Checking in GSTE

Jin Yang and Carl-Johan H. Seger

Strategic CAD Labs, Intel Corp.
{jin.yang,carl.seger}@intel.com

Abstract. We propose a compositional specification and verification approach based on GSTE (Generalized Symbolic Trajectory Evaluation). There are two main contributions. First, we propose a specification language that allows concurrent properties be described succinctly in a compositional algebraic manner. Second, we show a precise model checking solution for a compositional specification through automata construction, but much more importantly and practically, we develop an efficient model checking algorithm for directly verifying the compositional specification. At the end, we show the result of our approach in the verification of a micro-instruction scheduler in a state-of-the-art microprocessor.

1 Introduction

GSTE is a symbolic model checking solution that combines the high capacity and ease of use of STE with the expressive power (Ω-regular languages) of classic symbolic model checking [7, 22, 23]. It has been successfully applied to the formal verification of complex Intel designs with tens of thousands of state elements [22, 5, 21].

The specification language in GSTE is called *assertion graphs*, an operational formalism based on a special form of regular automata with assertion letters (antecedent and consequent pairs) as its input alphabet. Each word in the language of an assertion graph provides both a sequential stimuli for simulating the system and the expected sequential responses. The sequential nature of an assertion graph, however, hinders its ability to succinctly describe the concurrent behavior of a circuit.

In this paper, we present a compositional approach based on GSTE to overcome the limitation. First, we propose a specification language that allows the concurrent behavior of a system to be specified succinctly in a compositional manner. Such a composition is logical and does not rely on a deep understanding of the implementation details of the system. The language is an extension of the GSTE specification language with a new meet operator and is expressed in the form of Process Algebra [13, 19, 10]. Second, although we show that the compositional specification can be precisely model checked, we develop a much more efficient and practical solution to directly verify the compositional specification. The solution extends the GSTE model checking algorithm [22] with the ability to walk through the syntactical structure of the specification and establish a simulation relation from the language elements of the specification to the sets of states in the circuit. This avoids the exponentially expensive global assertion graph construction.

R. Alur and D.A. Peled (Eds.): CAV 2004, LNCS 3114, pp. 216–228, 2004.

There have been extensive studies on concurrent system specification and verification, most notably along the lines of hierarchical state machines (e.g. Statecharts) [9, 14, 4, 3, 2] and Process Algebra [19, 13, 10, 6]. However, most of these approaches have mainly focused on specification formalisms, correctness of specifications, and modular refinement strategies. None has provided an efficient and practical model checking solution to verify a concurrent specification against an implementation. Relying on model checking the global transition system for the specification is prohibitively expensive.

In recent years, the assume-guarantee based compositional approach has been gaining popularity [20, 19, 15, 8, 16–18, 11, 12], driven by the need to deal with the capacity limitation of symbolic model checking. In this framework, a circuit under verification is described as a parallel composition of finite state components, and the correctness of the circuit is described as a collection of local properties, each of which specifies the correctness of one component assuming that the correctness of the interfacing components. This approach achieves verification efficiency by model checking each local property against its component separately, and then establishing the global correctness using an inductive assume-guarantee reasoning. A main drawback of the approach, however, is that the specification is heavily implementation-dependent, requiring the deep understanding of how these components interact with each other. Therefore, it is rather manual, labor-intensive and sensitive to the changes in the implementation.

We firmly believe that a practical approach must contain two ingredients, a formal language to support succinct concurrent specifications, and an efficient solution (such as model checking) to verify such a specification against a complex implementation. Our GSTE based approach addresses both. The rest of the paper is organized as follows. In Section 2, we define assertion languages for GSTE and their trace semantics. In Section 3, we introduce the new binary *meet* operator $\sqcap$ for assertion languages. In Section 4, we present the compositional specification language for GSTE in a form of algebraic equations [13, 19, 10], and show that this language is well defined. We also prove a regularity result for a compositional specification, i.e., the limit of a repeated application of $\sqcap$ to any assertion language in the specification is regular. Since the limit is trace equivalent to the original language, this gives us a way to precisely model check the compositional specification using the GSTE model checking algorithm in [22, 23]. However, the construction of the regular automaton for the limit may cause an exponential blow-up in the size of the specification. Therefore in Section 5, we develop a GSTE model checking algorithm for directly verifying a compositional specification. In Section 6, we briefly discuss the result of the compositional GSTE approach in the verification of a micro-instruction scheduler from an Intel microprocessor design.

2 Assertion Languages

For the entire scope of the paper, we assume a finite, non-empty alphabet $\mathcal{D}$ called the *domain*. An ω-*trace* (or simply *trace*), denoted by $\tau = d_1 d_2 \ldots$, is any ω-word in $\mathcal{D}^\omega$. For a circuit under verification, the domain $\mathcal{D}$ is the set of all states in the circuit, and a trace is simply any infinite sequence of states.

We define the *assertion alphabet* Σ over domain $\mathcal{D}$ as the tuple $\Sigma = \mathbb{P}(\mathcal{D}) \times \mathbb{P}(\mathcal{D})$ where $\mathbb{P}(\mathcal{D})$ denotes the power set of $\mathcal{D}$. We call any letter in Σ an *assertion letter*, any word in Σ^* an *assertion word*, and any language in $\mathbb{P}(\Sigma^*)$ an *assertion language*.

Given an assertion letter $\sigma = (\mathcal{A}, \mathcal{B})$, $\mathcal{A}$ and $\mathcal{B}$ are called the *antecedent* and the *consequent* of the letter, denoted by $ant(\sigma) = \mathcal{A}$ and $cons(\sigma) = \mathcal{B}$. The antecedent and consequent functions are point-wise generalized to an assertion word w in Σ^*, i.e., $ant(w)$ and $cons(w)$ as defined by

$$\forall 1 \leq i \leq |w|, ant(w)[i] = ant(w[i]), cons(w)[i] = cons(w[i]).$$

Assertion graphs defined in [22, 23] are basically regular automata (or ω-regular automata if an additional fairness condition is specified) for generating regular assertion languages. We shall use the following simple *voting machine* example throughout the paper to illustrate various concepts in the paper.

Example 1. (Voting Machine) A voting machine (VM) consists of three voting stations (Figure 1). Each voting station receives a single vote through $vin[i]$ ($1 \leq i \leq 3$). Once all stations have received their votes, the VM conducts a poll among the three stations and produces a final voting result *vout*. It then clears all voting stations to accept a new round of votes. Output signal $av[i]$ tells if the i-th voting station is available. Signal *reset* resets the VM to its initial state. For simplicity, we ignore the content of a vote and that of the final result, and just care about control/status signals.

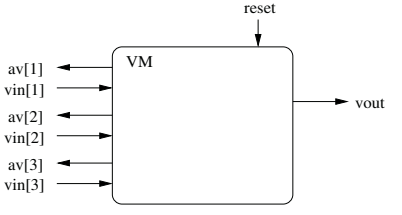

Fig. 1. Voting Machine (VM)

The domain D for the VM is the set of values for the external signals. For instance,

$$[reset = 0, vin[1] = 1, av[1] = 1, vin[2] = 0, av[2] = 0, vin[3] = 0, av[1] = 1, vout = 0]$$

is a value in D. We use state predicates over D to represent sets of values in D, for instance, $vin[1] \vee \neg vin[2]$ represents all values in D where either $vin[1] = 1$ or $vin[2] = 0$. The assertion word for the VM

$$((reset, true), (vin[1] \wedge \neg vin[2] \wedge \neg vin[3], true), (vin[2] \wedge vin[3], true), (true, vout))$$

says that after reset, if vote $vin[1]$ comes in followed by votes $vin[2]$ and $vin[3]$, then the VM produces the final result in the next time. □

Note that the specification in Figure 1 does not address how the system is implemented. As we shall show later in the paper, our compositional approach allows one to state the correctness of the system succinctly without the worry of implementation details. It empowers the model checking algorithm to connect the specification with the implementation of the system.

An assertion language is in some sense an operational specification formalism, where each assertion word in the language provides both a sequential stimuli for simulating the system and the expected sequential responses. Many or even an infinite number of words may be needed to cover all possible behaviors of the system. An assertion graph in [22, 23] is a way to use a labeled finite graph to capture theses words. However, it can still be quite a cumbersome way to describe a system with inherent concurrency. For instance, there are $1+3\times 2!+3!=13$ different orders for the three votes to arrive to the VM, each of which must be captured by a path in the assertion graph for the VM, although the specification does not care about in which particular order the votes arrive. The compositional extension in this paper addresses this issue.

In the following, we define the semantics of an assertion language. We say a trace τ over $\mathcal{D}$ *satisfies* a word π over $\mathbb{P}(\mathcal{D})$, denoted by $\tau\models\pi$, iff $\forall 1\leq i\leq|\pi|$, $\tau[i]\in\pi[i]$. We say τ *satisfies* an assertion word w in Σ^*, iff $\tau\models ant(w)\ \Rightarrow\ \tau\models cons(w)$. The *trace language* of an assertion word w, denoted $\Omega(w)$, is the set of all traces satisfying w, i.e.,

$$\Omega(w)=\{\tau\in\mathcal{D}^{\omega}\,|\,\tau\models w\}. \tag{1}$$

The *trace language* of an assertion language $\mathcal{L}$, denoted by $\Omega(\mathcal{L})$, is the *intersection* of the trace languages of the assertion words in $\mathcal{L}$, i.e.,

$$\Omega(\mathcal{L})=\cap_{w\in\mathcal{L}}\Omega(w). \tag{2}$$

This is the same as the semantics for assertion graphs in [22, 23]. Because of the $\forall$-semantics, the union $\cup$ of two assertion languages becomes stronger and yields fewer traces, in contrast to the traditional wisdom. The $\forall$-semantics is the basis for efficient GSTE model checking. The following theorem shows that a language with fewer assertion words yields more traces.

Theorem 1. $\mathcal{L}_1\subseteq\mathcal{L}_2\ \Rightarrow\ \Omega(\mathcal{L}_1)\supseteq\Omega(\mathcal{L}_2)$.

3 The Meet Operator

To facilitate compositional specifications, we introduce a *meet* operator $\sqcap:\Sigma\times\Sigma\rightarrow\Sigma$ that takes two assertion letters $\sigma_1,\sigma_2\in\Sigma$ and produces another assertion letter such that

$$ant(\sigma_1\sqcap\sigma_2)=ant(\sigma_1)\cap ant(\sigma_2),\ \text{and}\ cons(\sigma_1\sqcap\sigma_2)=cons(\sigma_1)\cap cons(\sigma_2). \tag{3}$$

The meet operator is applied point-wise to two words $w_1,w_2\in\Sigma^*$ of the same length, denoted by $w_1\sqcap w_2$, such that

$$w_1\sqcap w_2=\begin{cases}\varepsilon & \text{if } w_1=w_2=\varepsilon\\ (w_1'\sqcap w_2')\cdot(\sigma_1\sqcap\sigma_2) & \text{if } w_1=w_1'\cdot\sigma_1 \text{ and } w_2=w_w'\cdot\sigma_2,\end{cases} \tag{4}$$

where $\cdot$ is the language concatenation operator. It is not difficult to show that the operator is associative, commutative and idempotent. Finally, the meet operator is generalized to two languages $\mathcal{L}_1,\mathcal{L}_2\in\mathbb{P}(\Sigma^*)$:

$$\mathcal{L}_1\sqcap\mathcal{L}_2=\{w_1\sqcap w_2\,|\,w_1\in L_1,w_2\in L_2,|w_1|=|w_2|\}. \tag{5}$$

Example 2. Consider the VM specification in Example 1. Language ($1 \leq i \leq 3$)

$$\mathcal{V}[i] = (reset, true) \cdot (\neg vin[i], true)^* \cdot (vin[i], true) \cdot (\neg vin[i], true)^*$$

describes the first vote at the i-th station. The meet $\mathcal{V}_1 \sqcap \mathcal{V}_2 \sqcap \mathcal{V}_3$ succinctly describes all possible sequences of getting the first three votes without the explicit enumeration. □

In the following, we define a repeated application of $\sqcap$ over a language $\mathcal{L}$, $\sqcap^k \mathcal{L}$, as

$$\sqcap^k \mathcal{L} = \begin{cases} \mathcal{L} & \text{if } k = 0 \\ (\sqcap^{k-1} \mathcal{L}) \sqcap \mathcal{L} & \text{if } k > 0. \end{cases} \tag{6}$$

We can show that although the resulting meet language may produce new words, it is trace equivalent to the original language.

Lemma 1. *For all $k \geq 0$,*

1. $\sqcap^k \mathcal{L} \subseteq \sqcap^{k+1} \mathcal{L}$,
2. $\Omega(\sqcap^k \mathcal{L}) = \Omega(\sqcap^{k+1} \mathcal{L})$.

Now consider the limit $\bigcup_{m \geq 0} \sqcap^m \mathcal{L}$ for the ascending chain $(\sqcap^0 \mathcal{L}, \sqcap^1 \mathcal{L}, \sqcap^2 \mathcal{L}, \ldots)$.

Theorem 2.

1. *(Containment)* $\mathcal{L} \subseteq \bigcup_{m \geq 0} \sqcap^m \mathcal{L}$,
2. *(Trace Equivalence)* $\Omega(\mathcal{L}) = \Omega(\bigcup_{m \geq 0} \sqcap^m \mathcal{L})$.

The proof directly follows Lemma 1 by a transitivity argument. This result is important in establishing a regularity result for a compositional specification in Section 4.

4 Compositional Specification

We define a *compositional specification* $C(\Pi)$ as a set of algebraic equations over a set of assertion languages

$$\Pi = \{\mathcal{L}_0\} \cup \{\mathcal{L}_1, \ldots, \mathcal{L}_{h-1}\} \cup \{\mathcal{L}_h, \ldots, \mathcal{L}_{l-1}\} \cup \{\mathcal{L}_l, \ldots, \mathcal{L}_{n-1}\},$$

in which each equation is of the form

1. (Initialization)
$$\mathcal{L}_0 = \varepsilon \cup \mathcal{L}_0 \cdot \sigma_0, \tag{7}$$
where $\sigma_0 = (\mathcal{D}, \mathcal{D})$, or
2. (Prefix)
$$\mathcal{L}_i = \mathcal{L}_j \cdot \sigma_j, \tag{8}$$
for each $1 \leq i < h$ where $0 \leq j < n$ and $\sigma_j \in \Sigma$, or
3. (Summation)
$$\mathcal{L}_i = \mathcal{L}_{i_1} \cup \ldots \cup \mathcal{L}_{i_{k_i}}, \tag{9}$$
for each $h \leq i < l$ where $1 \leq i_j < h$ for $1 \leq j \leq k_i$, or

4. (Meet)
$$\mathcal{L}_i = \mathcal{L}_{i_1} \sqcap \ldots \sqcap \mathcal{L}_{i_{k_i}}, \tag{10}$$
for each $l \leq i < n$ where $q \leq i_j < l$ for $1 \leq j \leq k_i$,

where ε denotes the singleton language with the empty word $\{\varepsilon\}$ and $\cdot$ denotes the concatenation of a letter to the end of each word in a language.

This style of compositional definition is similar to Milner's CCS (*Calculus of Communicating Systems*) [19] with three differences: a special initialization equation, the meet operator in place of the parallel composition operator |, and (3) the trace semantics of assertion languages. Note also that this style is a generalization of assertion graphs in [22, 23]. In fact, without any meet composition, it corresponds to an assertion graph where (1) the initial language corresponds to the initial vertex with a self-loop, (2) a prefix language corresponds to an edge, and (3) a summation language corresponds to a vertex in the graph.

Example 3. The specification of the VM in Figure 1 is captured by the following set of algebraic equations, shortened by use a mixture of language and regular expressions.

1. (Ready) Station i ($1 \leq i \leq 3$) is in its $Ready[i]$ state after being reset or polled.
$$\begin{aligned} Ready[i] = (\,&(true, true)^* \cdot (reset, av[i]) \cup Poll \cdot (reset \vee \neg vin[i], av[i])\,) \\ &\cdot (reset \vee \neg vin[i], av[i])^*. \end{aligned}$$
2. (Voting) Station i ($1 \leq i \leq 3$) is accepting a vote.
$$Voting[i] = Ready[i] \cdot (\neg reset \wedge vin[i], av[i]).$$
3. (Vote) Station i ($1 \leq i \leq 3$) has got a vote.
$$Voted[i] = Voting[i] \cup (Voted[i] \sqcap Wait) \cdot (\neg reset, \neg av[i]).$$
4. (Wait) At least one voting station is in its *Ready* state.
$$Wait = \cup_{i=1}^{3} Ready[i].$$
5. (Poll) Every station is in its *Voted* state and one station is accepting a vote.
$$Poll = (\sqcap_{i=1}^{3} Voted[i]) \sqcap (\cup_{i=1}^{3} Voting[i]).$$
6. (Output) The VM outputs the polling result.
$$Output = Wait \cdot (true, \neg vout) \cup Poll \cdot (true, vout).$$

□

Note that this specification is not a conjunction of simple independent properties. Rather it is a network of semantically inter-dependent "communicating" properties. The following theorem shows that this set of equations is well defined in the algebraic sense.

Theorem 3. *The set of equations $C(\Pi)$ has a unique language solution.*

The proof is based on Tarski's fix-point theorem and an induction on the length of the words in the languages in the equations. We omit the proof due to the page limitation. In the following, we show that the limit of $\sqcap^m \mathcal{L}_i$ to each language $\mathcal{L}_i$ in the compositional specification is regular. This is significant based on Theorem 2, as it gives us a way to precisely model check the specification by constructing an assertion graph for the language and verifying the graph using the GSTE model checking algorithm in [22, 23]. To make the argument simple, we break each $\cup$ and $\sqcap$ composition into a series of pairwise compositions by introducing intermediate languages.

Lemma 2.

1. $\bigcup_{m\geq 0} \sqcap^m (\mathcal{L} \cdot \sigma) = (\bigcup_{m\geq 0} \sqcap^m \mathcal{L}) \cdot \sigma$.
2. $\bigcup_{m\geq 0} \sqcap^m (\mathcal{L}_1 \sqcap \mathcal{L}_2) = (\bigcup_{m\geq 0} \sqcap^m \mathcal{L}_1) \sqcap (\bigcup_{m\geq 0} \sqcap^m \mathcal{L}_2)$.
3. $\bigcup_{m\geq 0} \sqcap^m (\mathcal{L}_1 \cup \mathcal{L}_2) = (\bigcup_{m\geq 0} \sqcap^m \mathcal{L}_1) \cup (\bigcup_{m\geq 0} \sqcap^m \mathcal{L}_2) \cup (\bigcup_{m\geq 0} \sqcap^m \mathcal{L}_1) \sqcap (\bigcup_{m\geq 0} \sqcap^m \mathcal{L}_2)$.

The proof of the lemma is done by distributing $\sqcap$ over $\cdot$ and $\cup$, and then using the language containment argument. We omit the proof due to the page limitation.

Theorem 4. *$\bigcup_{m\geq 0} \sqcap^m \mathcal{L}_i$ is regular for every language $\mathcal{L}_i$ ($0 \leq i < n$) in Π.*

Proof. Consider the power set of the limit languages $\mathbb{P}(\bigcup_{m\geq 0} \sqcap^m \mathcal{L}_i \mid \mathcal{L}_i \in \Pi)$. Based on Lemma 2, each language in the power set can be expanded algebraically into either a prefix composition of another language, or a summation composition of some other languages in the set. Without the meet, the construction of a regular automaton for each language in the set is straight forward. Since $\bigcup_{m\geq 0} \sqcap^m \mathcal{L}_i$ is in the set, the claim holds. □

5 Direct Model Checking of Compositional Specification

Although model checking any language in the compositional specification can be done through the construction of the regular automaton for the meet limit of the language, such a construction is likely to cause an exponential blow-up in the specification size. To avoid the problem, we develop a GSTE model checking algorithm that directly walks through the syntactical structure of the specification and establishes a simulation relation from the language elements in the specification to the sets of states in the circuit.

We first define a *model* over $\mathcal{D}$ as a triple $M = (\mathcal{S}, \mathcal{R}, L)$ where

1. $\mathcal{S}$ is a finite set of *states*,
2. $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is a *state transition relation* over $\mathcal{S}$ such that for every state $s \in \mathcal{S}$, there is a state $s' \in \mathcal{S}$ satisfying $(s, s') \in \mathcal{R}$,
3. $L : \mathcal{S} \rightarrow \mathcal{D}$ is a *labeling* function that labels each state $s \in \mathcal{S}$ with a value $d \in \mathcal{D}$.

M induces a monotonic *post-image* transformation function $post_M : \mathbb{P}(S) \rightarrow \mathbb{P}(S)$:

$$post_M(\mathcal{S}') = \{s' \mid \exists s \in \mathcal{S}', (s, s') \in \mathcal{R}\}, \tag{11}$$

for all $\mathcal{S}' \in \mathbb{P}(S)$. We drop the subscript M when it is understood. A model is a Kripke structure without the initial state. To follow the STE convention, we assume that every state in M is an initial state. We extend L to state sets and define its inverse function L^-

$$L(S') = \{L(s) \mid s \in S'\}, \text{ and } L^{-}(D') = \{s \in S \mid L(s) \in D'\} \tag{12}$$

for all $S' \in \mathbb{P}(S)$ and $D' \in \mathbb{P}(D)$.

A *run* of the model is any mapping function $\gamma\colon \mathbb{N} \to \mathcal{S}$ such that for all $i \geq 0$, $(\gamma(i), \gamma(i+1)) \in \mathcal{R}$. The trace generated by γ, denoted by $L(\gamma)$, is the point-wise application of L over γ, i.e., $L(\gamma)(i) = L(\gamma(i))$ for all $i \geq 0$. The *trace language* of the model, denoted by $\Omega(M)$, is the set of all traces generated by the runs of the model, i.e., $\Omega(M) = \{L(\gamma) \mid \gamma \text{ is a run of } M\}$.

We say that the model M *satisfies* an assertion language $\mathcal{L}$, denoted by $M \models \mathcal{L}$, if

$$\Omega(M) \subseteq \Omega(\mathcal{L}). \tag{13}$$

M *satisfies* the set of equations $C(\Pi)$, denoted by $M \models C(\Pi)$, if $M \models \mathcal{L}_i$ for all $\mathcal{L}_i \in \Pi$.

The key idea of the model checking algorithm is to compute a simulation relation for each language in Π on M. A *simulation relation* is a mapping from each language $\mathcal{L}_i$ $(0 \leq i < n)$ to a state set $\mathcal{T}_i \in \mathbb{P}(S)$, such that for every state $s \in S$, $s \in \mathcal{T}_i$ if there is a word $w \in \mathcal{L}_i$ and a run γ in M such that

$$(1)\ L(\gamma) \models ant(w), \text{ and } (2)\ s = \gamma(|w|). \tag{14}$$

The simulation relation captures, for each language, the end of any run in the model satisfying the antecedent sequence of a word in the language. It has nothing to do with consequents. The importance of the simulation relation is stated in the following lemma.

Lemma 3. *$M \models C(\Pi)$, if for every prefix language $\mathcal{L}_i = \mathcal{L}_j \cdot \sigma_j$, $\mathcal{T}_i \subseteq cons(\sigma_j)$.*

Proof. First we prove that $M \models \mathcal{L}_i$ for every prefix language. Assume $M \not\models \mathcal{L}_i$. Then by (13) and (1), there is run γ of M and a word $w = w' \cdot \sigma_j$ in $\mathcal{L}_i$ such that (1) $L(\gamma) \models ant(w)$, but (2) $L(\gamma(|w|) \notin cons(\sigma_j)$. By (14), $\gamma(|w|)$ is in $\mathcal{T}_i$, and thus $L(\mathcal{T}_i) \not\subseteq cons(\sigma_j)$. Further, $M \models \mathcal{L}_0$ since $\mathcal{L}_0 = (\mathcal{D}, \mathcal{D})^*$. Based on this result, It also becomes obvious that $M \models \mathcal{L}_i$ for every $\cup$-composition $\mathcal{L}_i$ $(h \leq i < l)$ by (2). Finally, consider a $\sqcap$-composition $\mathcal{L}_i = \mathcal{L}_{i_1} \sqcap \ldots \sqcap \mathcal{L}_{i_{k_i}}$. Consider a trace τ in $\Omega(M)$. Let $w = w_1 \sqcap \ldots \sqcap w_{k_i}$ be a word in $\mathcal{L}_i$. By (13), we have $\tau \models ant(w_j) \Rightarrow \tau \models cons(w_j)$ for all $1 \leq j \leq k_i$. Now let us assume $\tau \models ant(w)$. Then by (4), $\tau \models ant(w_j)$ and therefore $\tau \models cons(w_j)$ for $1 \leq j \leq k_i$. Thus, $\tau \models cons(w)$. Therefore, $\tau \in \Omega(\mathcal{L}_i)$ by (5) and thus $M \models \mathcal{L}_i$. □

We now show how to iteratively compute a simulation relation for the specification based on its structure. Let $\mathcal{S}_n$ to denote the n-ary state set tuple $(\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{n-1})$. We define a partial order relation $\preceq\colon \mathcal{S}_n \preceq \mathcal{S}'_n$, iff $\forall 0 \leq i < n, \mathcal{S}_i \subseteq \mathcal{S}'_i$. $(\mathbb{P}(\mathcal{S}) \times \ldots \times \mathbb{P}(\mathcal{S}), \preceq)$ forms a finite c.p.o. with the bottom element being $\emptyset_n = (\emptyset, \ldots, \emptyset)$.

We define an update function for n-ary state set tuples on model M

$$Y(\mathcal{S}_n) = (Y_0(\mathcal{S}_n), Y_1(\mathcal{S}_n), \ldots, Y_{n-1}(\mathcal{S}_n)) \tag{15}$$

where

1. (Initialization) $\mathcal{L}_0 = \varepsilon \cup \mathcal{L}_0 \cdot \sigma_0$:

$$Y_0(\mathcal{S}_n) = S, \tag{16}$$

2. (Prefix) $\mathcal{L}_i = \mathcal{L}_j \cdot \sigma_j$ for $1 \leq i < h$:

$$Y_i(\mathcal{S}_n) = \begin{cases} L^-(ant(\sigma_j)) & \text{if } L_j = L_0 \\ post(\mathcal{S}_j) \cap L^-(ant(\sigma_j)) & \text{otherwise,} \end{cases} \tag{17}$$

3. (Summation) $\mathcal{L}_i = \mathcal{L}_{i_1} \cup \ldots \cup \mathcal{L}_{i_{k_i}}$ for $h \leq i < l$:

$$Y_i(\mathcal{S}_n) = \cup_{j=1}^{k_i} \mathcal{S}_{i_j}, \tag{18}$$

4. (Meet) $\mathcal{L}_i = \mathcal{L}_{i_1} \sqcap \ldots \sqcap \mathcal{L}_{i_{k_i}}$ for $l \leq i < n$:

$$Y_i(\mathcal{S}_n) = \cap_{j=1}^{k_i} \mathcal{S}_{i_j}. \tag{19}$$

It can be shown that Y is monotonic, and the sequence $(Y^0(\emptyset_n), Y^1(\emptyset_n), Y^2(\emptyset_n), \ldots)$ is an ascending chain with a least fixpoint, i.e., $\exists M \geq 0, \forall k \geq M, Y^k(\emptyset_n) = Y^M(\emptyset_n)$.

Lemma 4. *$Y^M(\emptyset_n)$ is a simulation relation for $C(\Pi)$ on M.*

The proof is based on an induction on the length of words leading to a state in the simulation relation, and is omitted due to the page limitation. Based on this result, we develop a GSTE model checking algorithm for the compositional specification in Figure 2. The following correctness result holds.

```
Algorithm: cGSTE(C(Π),post)
 1. T_0 := S, T_i := ∅ for all 1 ≤ i < n;
 2. active := {L_i | L_i = L_0 · σ_j};
 3. while active ≠ ∅
 4.       L_i := pickOne(active);
 5.       case L_i = L_0 · σ_j:          new := L^-(ant(σ_j));
 6.            L_i = L_j · σ_j:          new := post(T_j) ∩ L^-(ant(σ_j));
 7.            T_i = L_{i_1} ∪ ... ∪ L_{i_{k_i}}:  new := ∪_{j=1}^{k_i} T_{i_j};
 8.            else:                     new := ∩_{j=1}^{k_i} T_{i_j};
 9.       endcase
10.       if new ≠ T_i
11.            add to active every L_k having L_i in its definition;
12.       T_i := new;
13. endwhile
14. if T_i ⊈ cons(σ_j) for some L_i = L_j · σ_j
15.       return(false);
16. return(true);
end.
```

Fig. 2. cGSTE

Theorem 5. *$M \models C(\Pi)$, if $cGSTE(C(\Pi), M)$ returns true.*

The algorithm initially sets the simulation relation to the empty set for every language except for $\mathcal{L}_0$, which is set to $\mathcal{S}$. It then iteratively updates the simulation relation

for a language by locally propagating the simulation relations for the languages in its definition. It does so until no change can be made to any simulation relation. By avoiding the construction of a "global" automaton which may cause an exponential blow-up in the specification size, it becomes conservative but gains great efficiency.

To show the advantage of our approach over the assume-guarantee based compositional approach, we continue on the VM in Example 1. Figure 3 shows two different implementations of the VM. Implementation (1) is partitioned into four modules, one for each station and one for the final polling. Each station i tracks its own voting status in $v[i]$. Implementation (2) bundles all the signals from and to the three stations into 3-bit vectors, and the vector $av[1:3]$ tracks the availability status of each station. Assume that clr (set) sets the register to 0 (1) for the current and next steps.

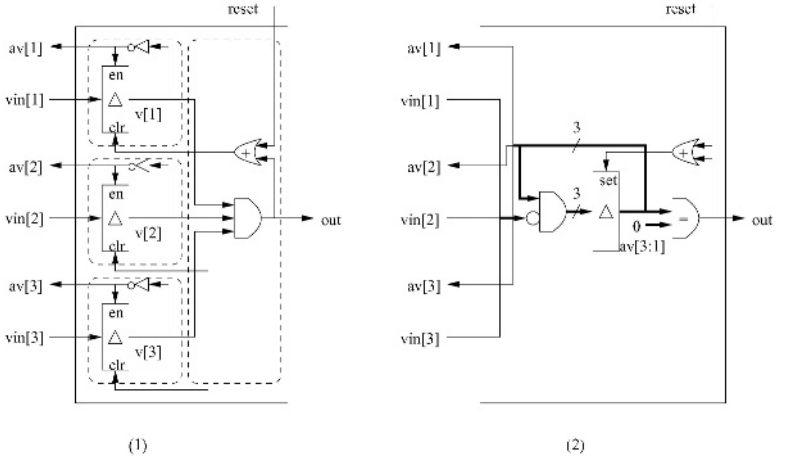

Fig. 3. Two VM Implementations

The assume-guarantee based approach is heavily implementation dependent and requires a clear understanding of the implementation details of the VM. For instance, given implementation (1), the overall specification may be decomposed into four local properties for the four modules, linking together through the interface behaviors of the voting status signals $v[1]$, $v[2]$ and $v[3]$. The property for the polling module may say, among other things, that if $v[i]$ is high for all $1 \leq i \leq 3$, then $out = 1$. The correctness of the decomposition also needs to be justified. Further, the change of the implementation could require an entirely different decomposition. For instance, the decomposition for implementation (2) would be bit-slicing based and relies on behaviors of the availability signals $av[3:1]$ to glue local properties together. It is conceivable that such a manual approach will be labor-intensive and difficult to get it right for complex designs.

On the contrary, our compositional specification is implementation independent. The languages in the specification can be viewed as a logical decomposition of the specification with no mentioning of the internal signal behaviors. Our model checking algorithm automatically computes the mapping from the end behavior specified by each language to the set of corresponding circuit states for any given implementation. Table 1 summarises the final simulation relations for $Ready[i]$, $Voting[i]$ and $Voted[i]$ computed

by the algorithm on the two implementations. Based on this, the simulation relation for *Poll* on implementation (1) is $\neg clr \wedge (\wedge_{i=1}^{3}(vin[i] \vee v[i])) \wedge (\vee_{i=1}^{3}(vin[i] \wedge \neg v[i]))$, which allows one to conclude that the implmentation will indeed satisfy $out = 1$ at the next step. Finally, we point out that the quaternary simulation aspect of GSTE allows our algorithm to focus only the relevant part of the model at each iteration step and store the simulation relations efficiently in an abstract form. We will not talk about it in the paper due to the page limitation.

Table 1. Final Simulation Relations for the VM

Language	Implementation (1)	Implementation (2)
$Ready[i]$	$(clr \vee \neg vin[i]) \wedge \neg v[i]$	$(set \vee \neg vin[i]) \wedge av[i]$
$Voting[i]$	$\neg clr \wedge vin[i] \wedge \neg v[i]$	$\neg set \wedge vin[i] \wedge av[i]$
$Voted[i]$	$\neg clr \wedge (vin[i] \vee v[i])$	$\neg set \wedge (vin[i] \vee \neg av[i])$

6 Verification of Micro-instruction Scheduler

The compositional GSTE has been implemented as a part of the GSTE verification system inside the Intel *Forte* environment ([1]). In this section, we discuss the verification of a micro-instruction (uop) scheduler in the original Intel®Pentium® 4 Microprocessor Scheduler/Scoreboard unit (SSU) (see Figure 4) as described in [5, 21]. The scheduler can hold and schedule up to 10 micro-instructions. The correctness property for the scheduler is that "when the resource is available, the oldest ready uop in the scheduler will be sent out." Even a much weaker version of the property, on the priority matrix module only stating "a uop would be scheduled if there are ready uops", had been quite difficult to prove previously using a state-of-the-art in-house symbolic model checker based on an assume-guarantee based approach. The logic involved created significant tool capacity problems, and required that the high level property be decomposed into literally hundreds of small local properties. Creating and proving this decomposition was done manually, and required a significant amount of time. Its maintenance and regression has been costly as the design changed.

Using the compositional GSTE approach, we were able to specify the entire correctness property succinctly in a compositional manner at the unit level, and verify it very efficiently using the compositional model checking algorithm. The compositional specification was developed in a top-down fashion. The top level property is

$$Prop = OldestReadyUop[i] \cdot (\neg stop, sched[i])$$

which simply says that if $uop[i]$ $(0 \leq i < 10)$ is the oldest ready uop, then schedule it at the next step if the resource is available. We then expand the property $OldestReadyUop[i]$:

$$OldestReadyUop[i] = Ready[i] \sqcap (\sqcap_{j \neq i}(NotReady[j] \cup EnqueuedEarlier[i, j]))$$

which simply defines the oldest ready uop as the uop that is ready and was enqueued to the scheduler earlier than any other ready uop. We keep expanding the properties until they are described in terms of input/output signal behaviors.

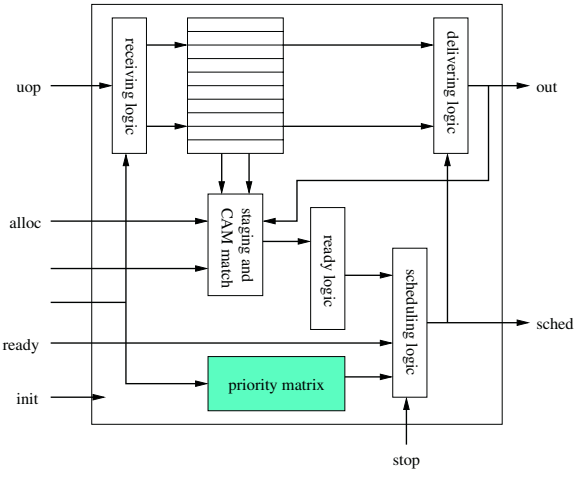

Fig. 4. Intel® Pentium® 4 Microprocessor SSU Scheduler

A recent rerun of the verification was done on a computer with 1.5 GHz Intel® Pentium® 4 processor with 1 GB memory. It was done on the original SSU circuit with no prior abstraction, which contains 718 latches, 361 inputs, and 17367 gates. The verification was done in 122 seconds using only 36MB memory. Most of the memory was used for storing the circuit. The tremendous verification efficiency benefits from the specification driven state space exploration strategy of the GSTE algorithm and the extended quaternary circuit abstraction technique [22]. Because of these, the verification complexity largely depends on the complexity of the specification to be verified rather than that of the circuit under verification, and is very scalable when the number of uops handled by the scheduler increases.

Finally, we would like to point out that compositional GSTE has been in active use in Intel and been successfully applied to the verification of several large-scale complex circuits in microprocessor designs.

7 Conclusion

In this paper, we presented a practical compositional extension to GSTE. For future work, we would like to extend the terminal condition and the fairness condition [23] to compositional GSTE, and apply the abstraction/refinement methodology in [22] to the compositional framework.

Acknowledgment

We would like to thank Ching-Tsun Chou, John O'Leary, Andreas Tiemeyer, Roope Kaivola, Ed Smith and reviewers for valuable feedbacks.

References

1. M. Aagaard, R. Jones, T. Melham, J. O'Leary, and C.-J. Seger. A methodology for large-scale hardware verification. In *FMCAD'2000*, November 2000.
2. R. Alur and R. Grosu. Modular refinement of hierarchical state machines. In *Proc. of the 27th ACM Symposium on Principles of Programming Languages*, pages 390–402, 2000.
3. R. Alur, R. Grosu, and M. McDougall. Efficient reachability analysis of hierarchic reactive machines. In *Computer-Aided Verification (LNCS1855*, pages 280–295, 2000.
4. R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. In *Proc. of the 26th International Colloquium on Automata, Languages, and Programming (LNCS1644)*, pages 169–178, 1999.
5. B. Bentley. High level validation of next generation micro-processors. In *IEEE International High-Level Design, Validation, and Test Workshop*, 2002.
6. J. Bergstra, A. Ponse, and S. Smolka. *Handbook of Process Algebra*. Elsevier, 2001.
7. C.-T. Chou. The mathematical foundation of symbolic trajectory evaluation. In *Computer Aided Verification*, July 1999.
8. E. Clarke, D. Long, and K. McMillan. A language for compositional specification and verification of finite state hardware controllers. *Proc. of the IEEE*, 79(9):1283–92, Sept. 1991.
9. D. Harel. A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
10. M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.
11. T. Henzinger, S. Qadeer, and K. Rajamani. You assume, we guaranee: Methodology and case studies. In *Computer Aided Verification (LNCS 1427)*, pages 440–451, 1998.
12. T. Henzinger, S. Qadeer, K. Rajamani, and S. Tasiran. An assume-guarantee rule for checking simulation. *ACM Trans. on Programming Languages and Systems*, 24:51–64, 2002.
13. C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
14. F. Jahanian and A. Mok. Modechart: A specification language for real-time systems. *IEEE Trans. on Software Engineering*, 20(2):933–947, Dec. 1994.
15. B. Josko. Verifying the correctness of aadl-modules using model checking. In *Proc. of the REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness (LNCS430)*. Springer, 1989.
16. D. Long. *Model Checking, Abstraction, and Compositional Reasoning*. PhD thesis, Computer Science Department, Carnegie Mellon University, 1993.
17. K. McMillan. A compositional rule for hardware design refinement. In *Computer Aided Verification*, June 1997.
18. K. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In *Computer Aided Verification*, June 1998.
19. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
20. A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, volume NATO ASI 13. Springer, 1997.
21. T. Schubert. High-level formal verification of next generation micro-processors. In *40th ACM/IEEE Design Automation Conference*, 2003.
22. J. Yang and C.-J. Seger. Generalized symbolic trajectory evaluation – abstraction in action. In *FMCAD'2002*, pages 70–87, November 2002.
23. J. Yang and C.-J. Seger. Introduction to generalized symbolic trajectory evaluation. *IEEE Trans. on VLSI Systems*, 11(3):345–353, June 2003.